JS函数的this指向

王红元 coderwhy

为什么需要this？

- 在常见的编程语言中，几乎都有this这个关键字（Objective-C中使用的是self），但是JavaScript中的this和常见的面向对象语言中的this不太一样：
  - 常见面向对象的编程语言中，比如Java、C++、Swift、Dart等等一系列语言中，this通常只会出现在类的方法中。
  - 也就是你需要有一个类，类中的方法（特别是实例方法）中，this代表的是当前调用对象。
  - 但是JavaScript中的this更加灵活，无论是它出现的位置还是它代表的含义。
- 我们来看一下编写一个obj的对象，有this和没有this的区别

```
var obj = {
  name: "why",
  running: function() {
    console.log(obj.name + " running");
  },
  eating: function() {
    console.log(obj.name + " eating");
  },
  studying: function() {
    console.log(obj.name + " studying");
  }
}
```

```
var obj = {
  name: "why",
  running: function() {
    console.log(this.name + " running");
  },
  eating: function() {
    console.log(this.name + " eating");
  },
  studying: function() {
    console.log(this.name + " studying");
  }
}
```

this指向什么呢？

- 我们先说一个最简单的，this在全局作用于下指向什么？
  - 这个问题非常容易回答，在浏览器中测试就是指向window

```
console.log(this); // window

var name = "why";
console.log(this.name); // why
console.log(window.name); // why
```

node环境没有window
node执行过程
modul->加载->编译->放到一个函数->执行这个函数.apply

- 但是，开发中很少直接在全局作用域下去使用this，通常都是在**函数中使用**。
  - 所有的函数在被调用时，都会创建一个执行上下文：
  - 这个上下文中记录着函数的调用栈、AO对象等；
  - this也是其中的一条记录；

this到底指向什么呢？

- 我们先来看一个让人困惑的问题：
  - 定义一个函数，我们采用三种不同的方式对它进行调用，它产生了三种不同的结果
- 这个的案例可以给我们什么样的启示呢？
  - 1.函数在调用时，JavaScript会默认给this绑定一个值；
  - 2.this的绑定和定义的位置（编写的位置）没有关系；
  - 3.this的绑定和调用方式以及调用的位置有关系；
  - 4.this是在运行时被绑定的；
- 那么this到底是怎么样的绑定规则呢？一起来学习一下吧
  - 绑定一：默认绑定；
  - 绑定二：隐式绑定；
  - 绑定三：显示绑定；
  - 绑定四：new绑定；

```
// 定义一个函数
function foo() {
  console.log(this);
}

// 1.调用方式一: 直接调用
foo(); // window

// 2.调用方式二: 将foo放到一个对象中,再调用
var obj = {
  name: "why",
  foo: foo
}
obj.foo() // obj对象

// 3.调用方式三: 通过call/apply调用
foo.call("abc"); // String {"abc"}对象
```

规则一：默认绑定

- 什么情况下使用默认绑定呢？独立函数调用。
  - 独立的函数调用我们可以理解成函数没有被绑定到某个对象上进行调用；
- 我们通过几个案例来看一下，常见的默认绑定

```
// 1.案例一:
function foo() {
  console.log(this);
}

foo();
```

window

```
// 2.案例二:
function test1() {
  console.log(this);
  test2();
}

function test2() {
  console.log(this);
  test3()
}

function test3() {
  console.log(this);
}
test1();
```

window

```
// 3.案例三:
function foo(func) {
  func()
}

var obj = {
  name: "why",
  bar: function() {
    console.log(this);
  }
}

foo(obj.bar);
```

规则二：隐式绑定

- 另外一种比较常见的调用方式是通过某个对象进行调用的：
  - 也就是它的调用位置中，是通过某个对象发起的函数调用。
- 我们通过几个案例来看一下，常见的默认绑定

```
// 1.通过对象调用
function foo() {
  console.log(this); // obj对象
}

var obj = {
  name: "why",
  foo: foo
}

obj.foo();
```

obj

```
function foo() {
  console.log(this);
}

var obj1 = {
  name: "obj1",
  foo: foo
}

var obj2 = {
  name: "obj2",
  obj1: obj1
}

obj2.obj1.foo();
```

obj1 (obj2.obj1 ==>obj1
obj1.foo())

```
function foo() {
  console.log(this);
}

var obj1 = {
  name: "obj1",
  foo: foo
}

// 讲obj1的foo赋值给bar
var bar = obj1.foo;
bar();
```

window

规则三：显示绑定

- 隐式绑定有一个前提条件：
  - 必须在调用的对象内部有一个对函数的引用（比如一个属性）；
  - 如果没有这样的引用，在进行调用时，会报找不到该函数的错误；
  - 正是通过这个引用，间接的将this绑定到了这个对象上；
- 如果我们不希望在 **对象内部** 包含这个函数的引用，同时又希望在这个对象上进行强制调用，该怎么做呢？
  - JavaScript所有的函数都可以使用call和apply方法（这个和Prototype有关
    - ✓ 它们两个的区别这里不再展开；
    - ✓ 其实非常简单，第一个参数是相同的，后面的参数，apply为数组，call为参数列表；
  - 这两个函数的第一个参数都要求是一个对象，这个对象的作用是什么呢？就是给this准备的。
  - 在调用这个函数时，会将this绑定到这个传入的对象上。

```
sum.call("call", 20, 30, 40)
sum.apply("apply", [20, 30, 40])
```

- 因为上面的过程，我们明确的绑定了this指向的对象，所以称之为 **显示绑定。**

call、apply、bind

- **通过call或者apply绑定this对象**
  - 显示绑定后，this就会明确的指向绑定的对象

箭头函数使用call绑定无效

```
function foo() {
  console.log(this);
}

foo.call(window); // window
foo.call({name: "why"}); // {name: "why"}
foo.call(123); // Number对象,存放时123
```

- **如果我们希望一个函数总是显示的绑定到一个对象上，可以怎么做呢？**

```
function foo() {
  console.log(this);
}

var obj = {
  name: "why"
}

var bar = foo.bind(obj);

bar(); // obj对象
```

内置函数的绑定思考

```
// 1.setTimeout
// function hySetTimeout(fn, duration) {
//   fn.call("abc")
// }

// hySetTimeout(function() {
//   console.log(this) // window
// }, 3000)

// setTimeout(function() {
//   console.log(this) // window
// }, 2000)
```

- 有些时候，我们会调用一些JavaScript的内置函数，或者一些第三方库中的内置函数。
  - 这些内置函数会要求我们传入另外一个函数；
  - 我们自己并不会显示的调用这些函数，而且JavaScript内部或者第三方库内部会帮助我们执行；
  - 这些函数中的this又是如何绑定的呢？
- **setTimeout、数组的forEach、div的点击**

```
setTimeout(function () {
  console.log(this); // window
}, 1000);
```

```
var box = document.querySelector(".box");
box.onclick = function () {
  console.log(this === box);
}
```

```
var names = ["abc", "cba", "nba"];
var obj = { name: "why" };
names.forEach(function (item) {
  console.log(this); // 三次obj对象
}, obj);
```

只有参数this指向window，
传入第二个参数，this指向第二个参数

```
var names = ["
names.forEach()
```

```
forEach(callbackfn: (value: string, index:
number, array: string[]) => void, thisArg?:
any): void

A function that accepts up to three arguments. forEach calls
the callbackfn function one time for each element in the array.
```

new绑定

- JavaScript中的函数可以当做一个类的构造函数来使用，也就是使用new关键字。
- 使用new关键字来调用函数是，会执行如下的操作：
  - 1.创建一个全新的对象；
  - 2.这个新对象会被执行prototype连接；
  - 3.这个新对象会绑定到函数调用的this上（this的绑定在这个步骤完成）；
  - 4.如果函数没有返回其他对象，表达式会返回这个新对象；

this是调用这个构造器时创建出来的对象
this = 创建出来的对象

```
// 创建Person
function Person(name) {
  console.log(this); // Person {}
  this.name = name; // Person {name: "why"}
}

var p = new Person("why");
console.log(p);
```

规则优先级

- 学习了四条规则，接下来开发中我们只需要去查找函数的调用应用了哪条规则即可，但是如果一个函数调用位置应用了多条规则，优先级谁更高呢？

- **1.默认规则的优先级最低**
  - 毫无疑问，默认规则的优先级是最低的，因为存在其他规则时，就会通过其他规则的方式来绑定this

- **2.显示绑定优先级高于隐式绑定**

  new>显式绑定(call,apply,bind)>隐式绑定>默认规则
  - 代码测试

- **3.new绑定优先级高于隐式绑定**
  - 代码测试

- **4.new绑定优先级高于bind**
  - new绑定和call、apply是不允许同时使用的，所以不存在谁的优先级更高
  - new绑定可以和bind一起使用，new绑定优先级更高
  - 代码测试

this规则之外 – 忽略显示绑定

- 我们讲到的规则已经足以应付平时的开发，但是总有一些语法，超出了我们的规则之外。（神话故事和动漫中总是有类似这样的人物）
- 如果在显示绑定中，我们传入一个null或者undefined，那么这个显示绑定会被忽略，使用默认规则：

```
function foo() {
  console.log(this);
}

var obj = {
  name: "why"
}

foo.call(obj); // obj对象
foo.call(null); // window
foo.call(undefined); // window

var bar = foo.bind(null);
bar(); // window
```

this规则之外 - 间接函数引用

- 另外一种情况，创建一个函数的 间接引用，这种情况使用默认绑定规则。
  - 赋值(obj2.foo = obj1.foo)的结果是foo函数；
  - foo函数被直接调用，那么是默认绑定；

属于独立函数调用

```
function foo() {
  console.log(this);
}

var obj1 = {
  name: "obj1",
  foo: foo
};

var obj2 = {
  name: "obj2"
}

obj1.foo(); // obj1对象
(obj2.foo = obj1.foo)();  // window
```

箭头函数 arrow function

- 箭头函数是ES6之后增加的一种编写函数的方法，并且它比函数表达式要更加简洁：
  - 箭头函数不会绑定this、arguments属性；
  - 箭头函数不能作为构造函数来使用（不能和new一起来使用，会抛出错误）；
- 箭头函数如何编写呢？
  - *(): 函数的参数*
  - *{}: 函数的执行体*

```
nums.forEach((item, index, arr) => {

})
```

箭头函数的编写优化

- 优化一: 如果只有一个参数()可以省略

```
nums.forEach(item => {})
```

- 优化二: 如果函数执行体中只有一行代码, 那么可以省略大括号
  - 并且这行代码的返回值会作为整个函数的返回值

```
nums.forEach(item => console.log(item))
nums.filter(item => true)
```

- 优化三: 如果函数执行体只有返回一个对象, 那么需要给这个对象加上()

```
var foo = () => {
  return { name: "abc" }
}
var bar = () => ({name: "abc"})
```

this规则之外 – ES6箭头函数

- 箭头函数不使用this的四种标准规则（也就是不绑定this），而是根据外层作用域来决定this。
- 我们来看一个模拟网络请求的案例：
  - 这里我使用setTimeout来模拟网络请求，请求到数据后如何可以存放到data中呢？
  - 我们需要拿到obj对象，设置data；
  - 但是直接拿到的this是window，我们需要在外层定义：var _this = this
  - 在setTimeout的回调函数中使用_this就代表了obj对象

```
var obj = {
  data: [],
  getData: function() {
    var _this = this;
    setTimeout(function() {
      // 模拟获取到的数据
      var res = ["abc", "cba", "nba"];
      _this.data.push(...res);
    }, 1000);
  }
}
```

ES6箭头函数this

- 之前的代码在ES6之前是我们最常用的方式，从ES6开始，我们会使用箭头函数：
  - 为什么在setTimeout的回调函数中可以直接使用this呢？
  - 因为箭头函数并不绑定this对象，那么this引用就会从上层作用于中找到对应的this

```
var obj = {
  data: [],
  getData: function() {
    setTimeout(() => {
      // 模拟获取到的数据
      var res = ["abc", "cba", "nba"];
      this.data.push(...res);
    }, 1000);
  }
}
```

```
var obj = {
  data: [],
  getData: () => {
    setTimeout(() => {
      console.log(this); // window
    }, 1000);
  }
}
```

- 思考：如果getData也是一个箭头函数，那么setTimeout中的回调函数中的this指向谁呢？

面试题一：

```
var name = "window";
var person = {
  name: "person",
  sayName: function () {
    console.log(this.name);
  }
};
function sayName() {
  var sss = person.sayName;
  sss(); // window
  person.sayName(); // person
  (person.sayName)(); // person
  (b = person.sayName)(); // window
}
sayName();
```

sss() 独立函数调用window
隐式调用persion
隐式调用persion
间接函数调用 window

coderwhy

面试题二：

```
var name = 'window'
var person1 = {
  name: 'person1',
  foo1: function () {
    console.log(this.name)
  },
  foo2: () => console.log(this.name),
  foo3: function () {
    return function () {
      console.log(this.name)
    }
  },
  foo4: function () {
    return () => {
      console.log(this.name)
    }
  }
}
```

```
var person2 = { name: 'person2' }

person1.foo1(); // person1
person1.foo1.call(person2); // person2

person1.foo2() // window
person1.foo2.call(person2) // window

person1.foo3()() // window
person1.foo3.call(person2)() // window
person1.foo3().call(person2) // person2

person1.foo4()() // person1
person1.foo4.call(person2)() // person2
person1.foo4().call(person2) // person1
```

coderwhy

面试题三：

```
var name = 'window'
function Person(name) {
  this.name = name
  this.foo1 = function () {
    console.log(this.name)
  }
  this.foo2 = () => console.log(this.name)
  this.foo3 = function () {
    return function () {
      console.log(this.name)
    }
  }
  this.foo4 = function () {
    return () => {
      console.log(this.name)
    }
  }
}
```

```
var person1 = new Person('person1')
var person2 = new Person('person2')

person1.foo1() // peron1
person1.foo1.call(person2) // person2

person1.foo2() // person1
person1.foo2.call(person2) // person1

person1.foo3()() // window
person1.foo3.call(person2)() // window
person1.foo3().call(person2) // person2

person1.foo4()() // person1
person1.foo4.call(person2)() // person2
person1.foo4().call(person2) // person1
```

coderwhy

面试题四：

```
var name = 'window'
function Person(name) {
  this.name = name
  this.obj = {
    name: 'obj',
    foo1: function () {
      return function () {
        console.log(this.name)
      }
    },
    foo2: function () {
      return () => {
        console.log(this.name)
      }
    }
  }
}
```

```
var person1 = new Person('person1')
var person2 = new Person('person2')

person1.obj.foo1()() // window
person1.obj.foo1.call(person2)() // window
person1.obj.foo1().call(person2) // person2

person1.obj.foo2()() // obj
person1.obj.foo2.call(person2)() // person2
person1.obj.foo2().call(person2) // obj
```

实现apply、call、bind

- 接下来我们来实现一下apply、call、bind函数：
  - 注意：我们的实现是练习函数、this、调用关系，不会过度考虑

```
Function.prototype.hyapply = function(thisBings, args) {
  thisBings = thisBings ? Object(thisBings): window
  thisBings.fn = this

  if (!args) {
    thisBings.fn()
  } else {
    var result = thisBings.fn(...args)
  }

  delete thisBings.fn

  return result
}
```

```
Function.prototype.hycall = function(thisBings, ...args) {
  thisBings = thisBings ? Object(thisBings): window
  thisBings.fn = this

  var result = thisBings.fn(...args)
  delete thisBings.fn

  return result
}
```

```
Function.prototype.hybind = function(thisBings, bindArgs) {
  thisBings = thisBings ? Object(thisBings): window
  thisBings.fn = this

  return function(...newArgs) {
    var args = [...bindArgs, ...newArgs]
    return thisBings.fn(...args)
  }
}
```

arguments

箭头函数没有argument

- **arguments 是一个 对应于 传递给函数的参数 的 类数组(array-like)对象。**

```
function foo(x, y, z) {
  // [Arguments] { '0': 10, '1': 20, '2': 30 }
  console.log(arguments)
}

foo(10, 20, 30)
```

- array-like意味着它不是一个数组类型，而是一个对象类型：
  - 但是它却拥有数组的一些特性，比如说length，比如可以通过index索引来访问；
  - 但是它却没有数组的一些方法，比如forEach、map等；

```
console.log(arguments.length)
console.log(arguments[0])
console.log(arguments[1])
console.log(arguments[2])
```